PREMIER MINISTRE

Secrétariat général
de la défense
et de la sécurité nationale

*Agence nationale de la sécurité
des systèmes d'information*

Paris, le XXX
N° YYY/ANSSI/SDE/PSS/CCN
Référence : ANSSI-CC-xxx-P-xx/i.j

WOOKEY SECURITY TARGET EVALUATION

NOTE: this is not an official security target.
This document has been produced for the **Inter-CESTI challenge**
(as a security target template on the test vehicle of the challenge)

51 boulevard de La Tour-Maubourg - 75700 PARIS 07 SP

Modifications logs

1.0	16/09/2019	Final version of the document
1.1	30/09/2019	Small fixes

Contents

1. Introduction

This document presents the WooKey security target evaluation, and as so it has a synthetic approach regarding the technical internals of the product. The curious reader can refer to the exhaustive online documentation [2] as well as the source code [3] for more details about the product rationale, architecture and implementation.

1.1. Product identification

The WooKey hardware revision is the publicly released **1.6 version**. The firmware version is the public stable release covered by the **manifest snapshot** `manifest_20190903.xml` file that can be found here [1]. This `manifest` file covers all the tagged versions of all the repositories needed to compile the firmware under evaluation.

2. Product description

WooKey is a self-encrypting USB device based on four hardware components:

- A motherboard PCB featuring a SoC (System-on-Chip) microcontroller embedding a dedicated firmware. This part will be called the *platform* in the sequel.
- A screen PCB and a screen that handle interactions of the en user with a screen (this PCB does not embed specific advanced algorithmic or firmware).
- A smart card (more specifically a Javacard) that embed a Secure Element (SE) whose purpose is to safely store the sensitive secrets at rest.
- An SD card that can be inserted/extracted from the platform. This SD card contains the user data in an encrypted from, ensuring *data at rest confidentiality*.

WooKey exposes to kinds of USB devices in its life-cycle corresponding to two operation modes:

- WooKey's *nominal mode* consists of exposing an **mass storage** USB device to a host, and transparently (de)encrypting data on the SD card after a user authentication. The platform must securely manage both the cryptographic and authentication materials along the user data path.
- The *DFU mode* is dedicated to secure firmware upgrades and exposes a **DFU class device**. This critical phase of the platform life cycle must be protected, and this will be explained how in the sequel of the document.

2.1. Product

2.2. Product hardware architecture

The architecture of WooKey is *open hardware*. The main motherboard and screen PCB layouts can be found here [4, 5].

2.2.1. WooKey main SoC: STM32F439

We focused on the `STM32F439` despite its lack of integrated High Speed PHY, as this SoC was the closest to our needs. This specific choice among the ST Cortex-M family has been further driven by two main aspects:

- On the software side, the STM32 Cortex-M4 SoCs have been widely studied in the recent years. Many efforts have been put in porting safe languages such as ADA and Rust to these processors.
- The `STM32F439` has a specific cryptographic acceleration coprocessor named STM32 hardware CRYP engine, which comes handy when dealing with encryption/decryption for high-speed data exchanges. A Hardware True Random Number Generator (TRNG) is also present.

Though this SoC has an integrated USB Full Speed PHY (12 Mb/s capable), it needs an external PHY to achieve High Speed (480 Mb/s). The communication between the SoC and the PHY is done using a ULPI interface, which is a standardized interface for USB 2.0.

2.2.2. Data storage

We have chosen to store the encrypted user data on external SD cards. They offer large storage capacities for an affordable cost with a possible expansion of the USB thumb drive capacity by switching the SD card. Compared to raw flash modules, there is no need to handle a complex FTL (Flash Translation Layer) since the firmware embedded in the SD card takes care of this. The SDIO (Secure Digital Input Output) bus only transfers encrypted data blocks on WooKey, which reduces the interest of compromising the SD card firmware and performing Man In The Middle attacks.

2.2.3. Touch screen

In order to limit the smartcard PIN code exposition and defeat Man In The Middle attacks on the USB bus or in a compromised host [11], we have decided to include a user input interface directly on the platform. This allows confining the PIN in the WooKey device. Among the possible input devices technologies, we have chosen the TFT-LCD `ILI9341` with a `AD7843` touch screen component. This allowed us to design a randomized PIN pad that makes movements observation attacks more complex [26]. We drive the touch screen from the STM32 SoC using the SPI bus where both the `ILI9341` and `AD7843` are slaves.

2.2.4. Authentication tokens: using the Javacard framework

A strong asset in the security approach of WooKey is the usage of an external authentication token. This allows to authorize data encryption and decryption operations and firmware updates only if a valid token is inserted and the user has provided the legitimate PIN codes using the touch screen.

We emphasize the fact that the external token does not solely serve a *logical presence* purpose: this critical element is actively used as a safeguard, through cryptographic operations, when a sensitive action is initiated. Using an external token with a strong user authentication allows us to exclude many attack scenarios where the USB device is lost by the legitimate user. If the hardware and software

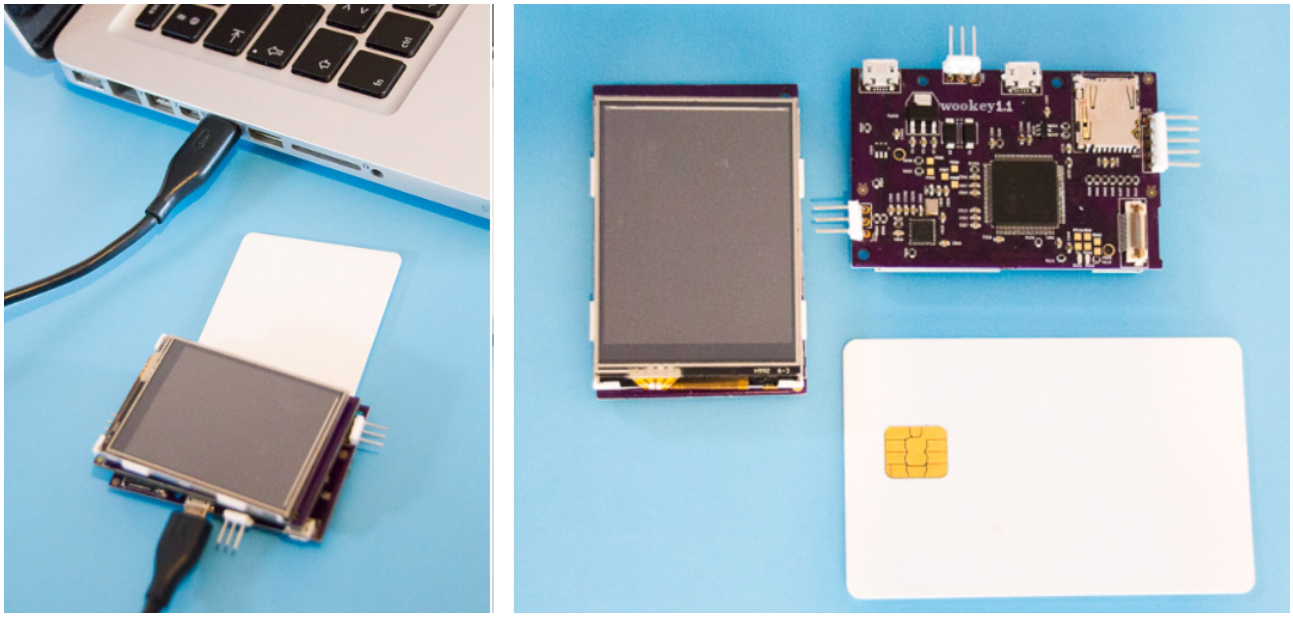

Figure 1: WooKey product overview

design are sound and if the decryption master secrets are stored in the token, an adversary will only be able to observe and abuse the pre-authentication modules of WooKey, and hence all the post-authentication critical modules are safe.

This however implies to **handle the cryptographic and authentication material securely**. This is the reason why we have decided to use a *secure element* to perform this task. Instead of soldering the secure element on the main board, an external authentication token is used (namely a smart card). this latter alternative has been chosen for security reasons: splitting the platform and the user token yields in a strong two-factors authentication scheme.

Finding a certified secure element that can be programmed without having NDAs and buying large quantities is not an easy task. This is particularly true if one wants to have a bare-metal access to the chip, e.g. to implement its own OS.

The attempts to bring secure elements technology to the public domain have emerged through VM-backed languages. The user code is confined in a Virtual Machine and the resources of the platform are abstracted with standard and documented APIs. This isolation serves two purposes. First, the low-level layers are protected against tampering, and the isolated applications cannot interfere with each other. Secondly, since the Virtual Machine API is standardized, there is no need to access low-level proprietary information to implement useful algorithms.

Javacard is the only *widely available framework* to offer a Common Criteria certification, thus we have chosen to focus on this platform. More specifically, we have developed and tested our applet on EAL 4 certified NXP JCOP `J3D081 2.4.2` smartcards [27]. The applets do not make use of any proprietary API, and should be compatible with all **Javacard 3.0.1** compliant cards.

2.3. Product usage description

WooKey must be plugged into a USB host device interface (the host device is usually a laptop or a working station PC). The end user must introduce his smart card in the dedicated connector on WooKey in order to unlock the device after an authentication sequence.

The authentication sequence is composed of the following steps:

1. When the smart card is detected in the dedicated connector, a pin pad is shown to the end user on the screen.

2. A first PIN (called the *PetPIN*) is introduced by the user on the touch screen. This PetPIN is used to initiate an ECDH mutual authentication between the device SoC and the smart card through a key derivation. The smart card allows only a limited number of bad PetPINs, and locks/self-destroys itself when this number is reached.

3. If the PetPIN is correct, a secret sentence called the *PetName* and stored inside the smart card is sent through the secure channel to the WooKey device. This sentence is printed on the screen, and the user is asked to compare it to the one he configured at device setup time, and to validate or decline it.

4. If the user validates the PetName, he is asked to provide a new PIN called the *UserPIN*. This PIN fully unlocks the smart card when it is correct (or locks/self-destroys the smart card after a number of bad PINs is reached).

5. When the smart card is fully unlocked with the UserPIN, the platform asks for the *master secrets* that are used to decrypt the user data on the SD card. Any attempt to remove the smart card will make these secrets erased and the platform reboot.

2.4. Product software architecture

2.4.1. WooKey general architecture overview

Classical USB thumb drives need at least two main software components: the USB stack to exchange data with the host and the mass storage manager to store data. One of WooKey main features is to encrypt the data at rest, which requires a dedicated cryptographic module to encrypt/decrypt this data. WooKey must securely manage both the cryptographic and authentication materials along the user data path.

The *data path* goes through three logical modules to read and write data from/into the device:

1. The USB module handles the USB communication with the host.

2. The SD module manages the mass storage device and read/write of encrypted data.

3. An untrusted cryptographic module: it shares memory space with the USB and SDIO tasks, and its job is to trigger encryption and decryption in the underlying CRYP hardware module (the CRYP module is the dedicated AES coprocessor inside the STM32 F439 MCU).

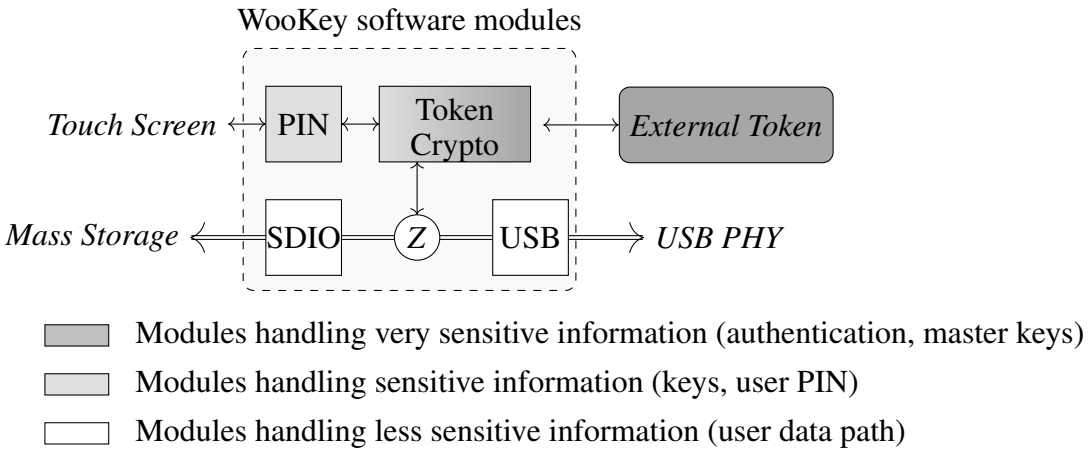

Figure 2: General software architecture of WooKey

The data path is isolated from the *authentication path*: this last path involves two software modules:

1. A trusted cryptographic module: this module is confined and isolated from the other tasks. It is in charge of setting up the CRYP key registers with the secret AES key derived from the external authentication token. It is also in charge of managing all the communication with this token (through an ISO7816 driver).

2. The PIN module that is in charge of user GUI (Graphic User Interface) and interactions through the touch screen, such as PIN input and PetName printing.

2.4.2. Defense in depth strategy

We retained several security requirements to bring to the WooKey platform a near state-of-the-art security level. In the first place, we use the MPU to protect the most sensitive assets while enforcing the least privilege principle. A dedicated *microkernel* is responsible for configuring the MPU. This microkernel is also implemented in a *safe language* for reasons described in 2.4.4..

Advanced in-depth *mitigation mechanisms* are also used. Stack canaries with proper randomness are configured by the compilation toolchain to limit stack overflows. In each task, the stack (lower) and the heap (upper) limits are guarded by dedicated MPU regions. Heap overflow protections (allocation randomization, heap canaries and consistency checks) are also implemented.

W⊕X is enforced by the microkernel: non-executable data regions and non-writable text regions ensure that no data payload harnessed by the attacker becomes directly exploitable (leaving only *Return-oriented* and *Jump-oriented* programming gadgets as a potential threat). The W⊕X is proven by a formal framework discussed in 2.4.5..

Unfortunately, the small RAM size and the lack of virtual memory rules out any useful ASLR (Address Space Layout Randomization) due to a shoestring entropy.

2.4.3. EwoK microkernel

All the software modules are isolated thanks to the *EwoK microkernel*. Most of modern 32-bit microcontrollers have a Memory Protection Unit and a processor with at least two CPU privilege levels (the so-called *user mode* and *supervisor mode*). The MPU is a programmable unit that allows privileged software, often a *kernel*, to define memory access permissions in order to isolate memory regions. EwoK uses this mechanism to enforce confinement and privilege separation between *tasks* executed in *user mode*, so that they cannot break out of their address space.

Implementing versatile memory isolation using the MPU is a real challenge because of its inner limitations: only 8 memory regions are simultaneously allowed, and strict alignment constraints must be fulfilled [19]. Such limitations explain why EwoK currently cannot execute more than 8 user tasks, which is enough for the WooKey use case and many embedded contexts.

WooKey makes use of the following features in order to bring a balanced (yet strict) security versus performance trade off:

- *Access control to devices and resources*: Access by a task to any hardware resource (device, timer) or software resource provided by the kernel (IPC, shared memory) is managed by a fixed and immutable set of permissions. To avoid unexpected side effects, declaration and setup of these resources can only be done during a first *initialization phase*, after which requesting further resources is prohibited for the tasks. Registered devices are mapped by the MPU in the tasks' address space. A difficulty arises when a user mode driver claims a hardware component that must be shared with other tasks.

  For example, when a task requires access to a specific GPIO (General Purpose Input/Output) port/pin, it would be dangerous to map the whole GPIO port registers in its address space as it would interfere with devices controlled by some other tasks. We found that the proper solution here is to use specific syscalls so that the kernel can strictly control the access to the GPIO pins.

- *DMA*: An efficient implementation requires using DMA (Direct Memory Access) to avoid slow byte-per-byte copies of large amounts of data. Nonetheless, performance comes at a cost: the DMA engine bypasses the MPU protection, putting at risk the whole system. Hence, controlling the DMA transfers is a challenge solved in EwoK by introducing specific syscalls. These allow the kernel to check that DMA requests are conforming to the security policy in place. Only DMA transactions from memory to devices and devices to memory are allowed. Due to their inherent pitfalls, memory to memory transactions are forbidden.

- *Fast interrupts acknowledgement*: Some hardware devices such as the smart card generate interrupts that must be acknowledged within a very tight time frame to avoid timeouts. Other components like the touch screen put pressure on the kernel with interrupts bursts. To deal with these constraints, we designed a simple yet effective system to quickly acknowledge interrupts and to limit as much as possible the overhead of the user mode drivers.

- *Posthook mechanism*: Posthook instructions define a restricted high level language that allow to read or to set some bits in specific hardware registers when an interrupt occurs. For each kind of interrupt, a driver can use such *posthook* instructions, that are synchronously interpreted and

executed by the kernel, in order to quickly acknowledge hardware interrupts. Posthooks are an interpreted boolean logic, not executable code.

- *IPC*: Tasks may communicate using synchronous or asynchronous IPC. The inter-tasks IPC permission matrix is statically defined at build time to avoid improper access or information leakage.

- *Resource release*: For least privilege enforcement, a specific syscall is dedicated to definitely *release resources*. Devices are then fully unmapped from a task's memory space.

2.4.4. Safe languages: using Ada

Most of kernels are written in C with some assembly. The major drawbacks of the C language are its proneness to coding errors. Out-of-bound arrays access, integer overflows and dangling pointers are difficult to avoid due to the weakly enforced typing. Such bugs might be devastating when exploited in a privileged context.

EwoK uses Ada, a language often chosen for building high-confidence and safety-critical applications [25, 8]. It is a strongly typed language that supports bare-metal programming and that enforces type checking both at compile time and at run time. Hence, more than 80% of the critical vulnerabilities (as per studies) could be avoided thanks to features inherent to this language.

2.4.5. Formal methods

Formal methods allow to prove the correctness of a design and/or of an implementation with respect to some predefined properties using mathematically based techniques. This approach fits well with microkernels.

We use SPARK [7] and the *GNATprove* tool [6] to prove that the kernel is free of run-time errors. It would provide great ensurance of the kernel robustness, and it would allow to remove Ada run-time checks, which would slightly improve the kernel performance.

Because of the impossibility to use Ada access types (similar to C pointers) with SPARK, only some kernel components have been proven using *GNATprove*. Hence, we have focused the usage of SPARK on components dealing with crucial security properties: among other assets, we prove that EwoK configures the MPU in a way that *always enforces the* $W \oplus X$ *paradigm*. We finally emphasize the fact that modules that are not proven in SPARK still benefit from Ada's properties, including *runtime checks*.

2.4.6. User data confidentiality

Full-Disk Encryption (FDE) has become a matter of concern and a topic of interest in applied cryptography these last years. The high level features an end user expects are both data *confidentiality* and *integrity*. Unfortunately, no ideal efficient solution exists nowadays since integrity expects extra data to be stored on the disk. This explains why most of FDE solutions only focus on user data confidentiality, and this is also the case for WooKey.

We have decided to use AES-CBC-ESSIV [15] (e.g. used in Android FDE [17]) because of performance reasons: the CBC mode is accelerated by the `CRYP` coprocessor of the `STM32F439`.

Although tweakable modes such as AES-XTS [14] are more popular and more resistant against block malleability [22], we stress out that integrity is still at risk. With WooKey, we clearly state that *integrity is not ensured* when a device or an SD card is lost: a straightforward solution for the end user is to handle it in a higher layer (e.g. file system).

2.4.7. External tokens and user authentication

The smart card extractable tokens are a cornerstone of WooKey's security. Since they are based on EAL certified chips, they are entrusted with the sensitive secrets, e.g. the user data at rest AES-CBC-ESSIV key and other assets.

Mutual authentication and secure channel: The main purpose of the cryptographic architecture that we describe in this section is to protect the WooKey device from *pre-authentication attacks*. That is to say, an attacker having access to the device but with only one of the two authentication factors (the token or the user PIN) will not be able to recover sensitive assets. The main platform and the external token are strongly bound thanks to a mutual authentication. The main SoC and the token embed personalized ECDSA authentication key pairs, yielding in an authenticated ephemeral ECDH (Elliptic Curve Diffie-Hellman) to derive AES-CTR, HMAC-SHA-256 session keys as well as a random IV (Initialization Vector) value. This establishes a session with a secure channel over the ISO7816 physical line with *confidentiality*, *integrity* and *anti-replay* properties. Forcing a mandatory mutual authentication mitigates man-in-the-middle adversaries, and limits the attack surface against malicious tokens and malicious ISO7816 masters.

Rogue tokens, PetPIN, PetName and UserPIN: When considering our threat model, an adversary could *steal the user PIN*. The scenario is the following: the attacker first steals both the platform and the token from the user while replacing them with ersatz in order to deceive the user[1]. When the legitimate user enters the PIN and realizes that the device is fake, it is too late since the PIN might have been sent over-the-air. In order to thwart such attacks, we use a two steps authentication involving two PIN codes: the PetPIN and the UserPIN as presented in Stage 2 of Fig. 5. The PetPIN partially unlocks the token while providing it along with the UserPIN fully unlocks it (to get sensitive secrets). When providing the PetPIN, the token sends back the PetName: this is a secret sentence that has been provisioned during setup by the user. This PetName is printed on the device screen allowing the user to check it and decide knowingly to enter his UserPIN, thus impeding rogue tokens scenarios.

STM32 assets protection: Although sensitive assets are safe inside the smart card secure element, this is less the case in the `STM32F439` SoC internal flash. Hardware flash readout protections are not bulletproof against adversaries performing fault attacks. This means that the platform ECDSA keys are at risk when the device is lost. In order to protect such keys, we encrypt them using a key derived from the PetPIN as represented in Stage 1 of Fig. 5. A straightforward – yet unsafe – way of doing this is to use a standard Key Derivation Function such as PBKDF2 [23, 24]. This is risky since the `STM32F439` has not enough power to support the number of iterations recommended against brute force attacks [13], knowing that users usually encode their PINs on few digits. We deal with offline exhaustive search by making the external token derive the ECDSA assets decryption key from the PetPIN PBKDF2 derived value: the adversary will need the slow and secure external hardware that severely restricts brute force attacks.

[1] The open source and hardware aspects of the project facilitate it.

AUTH, DFU and SIG tokens: For the sake of security, we have decided to dedicate a different token for the three main phases of the product. The AUTH token is used during the *nominal mode* and stores the data at rest AES-CBC-ESSIV master key. The DFU token is committed to the *DFU mode* and is in charge of managing the firmware decryption sessions keys (more on this in the dedicated section 2.4.9.). Finally, the SIG token is not directly used with the device *per se*: it is specifically devoted to protect the ECDSA firmware signature private key, derive encryption keys, and is used on the firmware production platform (e.g. a PC). The three tokens use the two stages user authentication and secure channel mounting protocol presented on Fig. 5 with dedicated ECDSA keys, PetPIN, PetName and UserPIN for each one.

2.4.8. Nominal mode software design

This mode of operation is composed of five *isolated user mode tasks*, each one handling one peripheral of the platform as presented on Fig. 3.

User data path: The USB module handles the USB stack to communicate with the host through SCSI [10] commands. The SD software module manages the mass storage device on the SDIO bus. The crypto module sits between these two modules, and drives the `CRYP` coprocessor. These three modules are dedicated to the *data path*: user data is transparently (de)encrypted along this path once the user is authenticated. In order to optimize the data flow, two *shared DMA* buffers are declared by the USB and the SD tasks, through dedicated syscalls, to be used by the crypto task as sources or targets for DMA transfers. Hence, the crypto task is able to program DMA transactions between the USB and the SD module via the `CRYP` device, allowing transparent data packets (de)encipherment.

Authentication path: Interestingly, although the crypto task manages the data path, it has never access to the storage *master key*: it only uses the `CRYP` device as a (de)encryption engine. All the platform *sensitive secrets* follow an *authentication path* that is completely separated from the mass storage data path. This ensures a defense in depth property for WooKey: compromising any of the exposed USB, SD or crypto tasks will not lead to critical assets leakage.

Two other software modules (smart card and PIN tasks) are devoted to the *authentication path*. The PIN task interacts with the touchscreen: it sends the PetPIN/UserPIN to (and gets the PetName from) the smart card task using IPCs. The smart card module handles the AUTH token, dealing with the ISO7816 layer and the secure channel, and gets the AES-CBC-ESSIV master key after a successful user authentication. This key is injected in the `CRYP` dedicated memory mapped area (only accessible to this task), allowing the crypto task to drive ciphering operations without knowing it.

2.4.9. DFU mode software design

Since firmware updates are usually the Achilles heel of embedded devices security, we have put some efforts to have a flexible, robust and secure upgrade process through a dedicated *DFU mode* of WooKey.

Flexibility comes from the usage of the Device Firmware Update protocol as standardized by the USB consortium [18]. This allows us to be compatible with existing classic tools.

Robustness is not so easily achievable because such devices are often not self-powered and may be disconnected at any time. We present how a flip-flop design reaches such a goal. A first – yet insufficient – fence against attacks is to use a dedicated *button* on the board to trigger the DFU mode

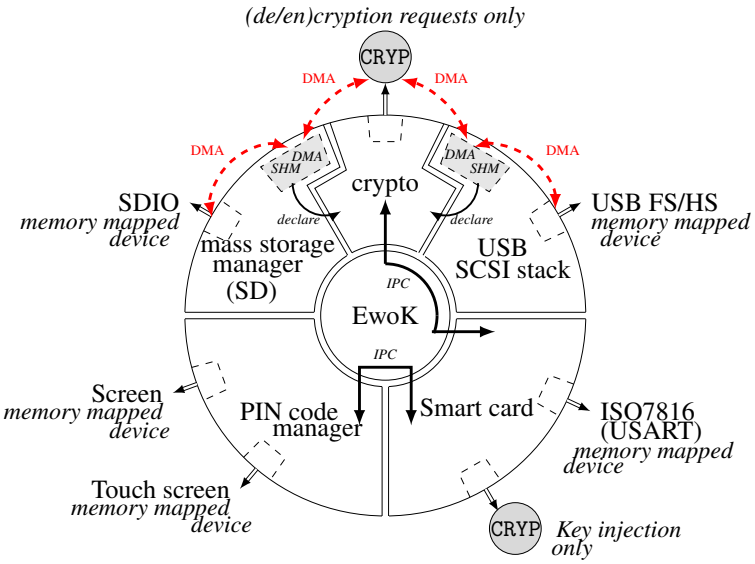

Figure 3: Software architecture – WooKey *nominal mode*

only with physical access. This thwarts remote attacks targeting unsolicited upgrades. Security is ensured by cryptographic guarantees as well as defense in depth using our microkernel.

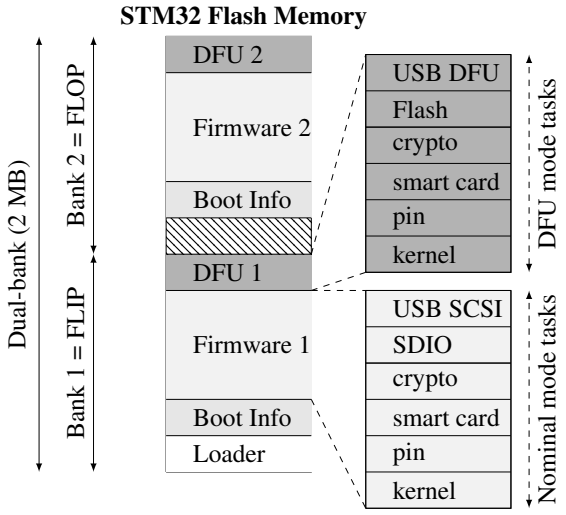

Figure 4: Overview of WooKey flip-flop layout

Flip-flop design: Because MCUs have a quite limited volatile memory, firmware upload and verification have to be performed in-place in the flash area where it will be executed. This inclined us to adopt a *flip-flop mechanism* ensuring software redundancy in order to handle file corruption (hazardous disconnection, corruption, invalid signature, etc.).

Fig. 4 provides a high level logical view of the flash layout. The 2 MB *dual-bank* of the `STM32F439` SoC internal flash is split in two. The first bank consists of the flip partition. It con-

tains the initial loader, some boot information, Firmware 1 which encapsulates the kernel and the tasks of the nominal mode, and DFU 1 that contains the kernel and the tasks of the DFU mode. The second bank is a replica of the first one with a mirrored layout containing a different version for Firmware 2 and DFU 2. The advantage of dual-banking is that a bank (the one being executed) can be write-protected with hardware ensurance, while the other bank is being updated.

The Boot Information section contains the current state of the firmware in the bank, namely a version number, a flag indicating if the last update has been consistently achieved, and a SHA-256 hash value to be checked by the initial loader. This loader is not upgradable but is *very minimal* with no I/O interactions (except for the DFU button). Since downgrading can be a boon for the adversary [9], strict *anti-rollback* is enforced both during the upgrade phase and at boot time.

Firmware signature and encryption: In order to ensure the firmware authenticity, we apply an ECDSA signature with a private key enclosed in the SIG token on a trusted dedicated host. A straightforward way of implementing the signature verification is to embed the ECDSA public key in the WooKey platform and check the signature after a firmware is written in flash (writing the firmware before checking it is unavoidable because of a very limited embedded RAM size). The flag in the boot information sector is flipped to a proper value if and only if this check is consistent. Since we want strong user authentication, the DFU token is used along with the PINs to validate the legitimate user presence.

Such a strategy suffers from two major drawbacks. First, the DFU token is uncorrelated to the update procedure (it is only used for access control), meaning that time of check to time of use (TOCTOU) attacks are possible. Secondly, this process is inherently susceptible to *fault attacks*. Indeed, a voltage glitch or an EM pulse performed at the right timing on the STM32 [16, 21, 12] could completely bypass the signature check, yielding in a malleable binary in flash and a full privileged compromise of the platform with another fault at boot time. As we have already stated, secure elements of the tokens are on the other hand protected against faults.

To limit such fault attacks, we use actively the DFU token during the whole update process as an oracle to derive session keys for firmware decryption using a dedicated enclosed secret key. Since the firmware is deciphered on-the-fly using keys unknown to the attacker, the data in flash is still malleable but its value is now *not controlled* by the adversary. Fig. 6 illustrates how the platform opens a session with the token and asks for key derivation to handle successive chunks. As we can see on the figure, we have designed a dedicated simple file format for update binaries. It consists of a header HDR followed by a body of encrypted chunks. The header is composed of metadata regarding the file (total size, version, chunks size, etc.), the ECDSA signature, an IV (initial value to produce keys) and HMAC-SHA-256 of $\overline{\text{HDR}}$ (i.e. the header except the HMAC itself). The signature covers the metadata and the firmware binary in clear (since we have to check this signature after writing clear data in flash). To avoid any padding related issue, we use an AES-CTR mode for firmware ciphering. The rationale behind the HMAC is to avoid malleability of the header and to early prevent opening illegal sessions with the token (solely counting on the signature implies a late detection). We also do not want the adversary to use the DFU token as an oracle to produce keys for any IV value (only the SIG token produces this HMAC).

It is fair to say that if an attacker is able to control one of the session keys values through a fault (e.g. by zeroing it), he will get back the control on data written to flash. He would still need to perform a fault on the signature check as well as on the hash check on boot, and more importantly to bypass the

DFU token authentication and secure channel. This theoretical multi-faults and multi-bypass scenario seems quite complex to achieve.

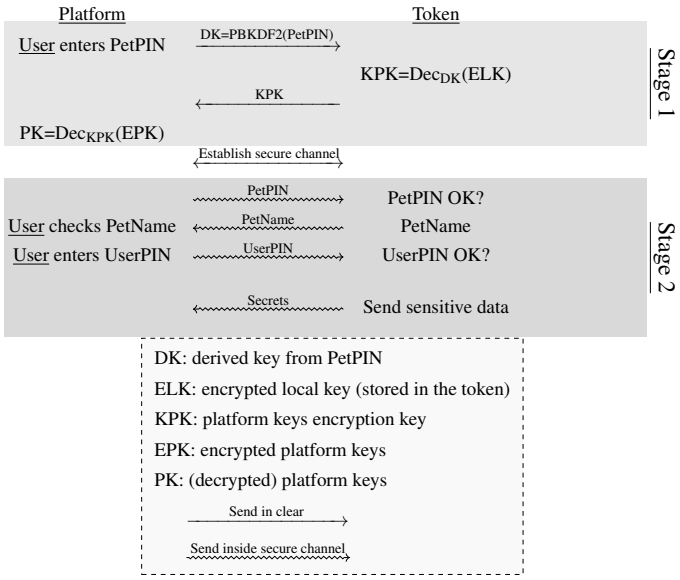

Figure 5: Sensitive data protection in WooKey

DFU mode defense in depth: As for the nominal mode of WooKey, we want the DFU mode to be protected against software attacks since a history of exploited vulnerabilities in such mode exists [20, 28]. Hence, we adopt the same defense in depth approach described in 2.4.8. using five isolated tasks above the microkernel and software mitigation (see Fig. 7).

The USB task implements the DFU standard, the flash manager writes the chunks in non-volatile memory, and the crypto task configures the DMA requests to and from the `CRYP` engine for transparent decryption. The smart card module handles the user authentication with the DFU token (with the PINs provided by the dedicated task), and manages the session keys derivation. With new firmware chunks, associated keys are derived by the token and injected in the `CRYP` registers. When the firmware decryption is over, the flash manager task *definitely releases* the flash device. This ensures that the signature can be verified by the smart card task using the `HASH` engine, and the Boot Information section can be atomically updated, without time of check to time of use issues.

2.5. Product life-cycle

The WooKey device has two main phases:

1. **The provisioning phase**: this is the early phase of the product. The WooKey main board is in RDP 0 (not locked), and can be programmed through JTAG/SWD. The WooKey AUTH/DFU/SIG tokens are unlocked with default GlobalPlatform keys. In this phase, the user compiles an initial firmware choosing the **production profile** (where the USART is disabled and full protections in the bootloader are active). After firmware compilation, the user flashes it in the board, and uses a third party tool to migrate the board to RDP2 (Readout Protection level 2). Moreover,

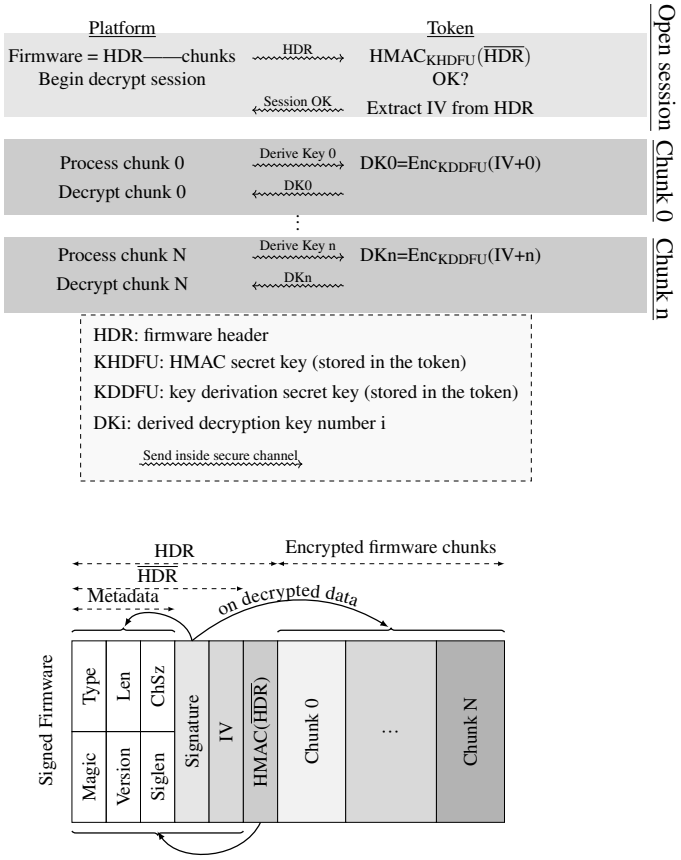

Figure 6: DFU cryptography protection in WooKey

the user programs the AUTH/DFU/SIG tokens with the respective applets and locks the GlobalPlatform keys with a third party tool. The user saves the generated `private` folder for future key recovery or token programming, along with his new GlobalPlatform keys: these sensitive data must be kept in a **place**. All these steps must be performed in a **secure environment** using a **disconnected and trusted PC host**.

2. **The final usage phase**: the WooKey device and tokens are now fully configured and can be deployed in unsafe environments. Whenever it is necessary to update the device, the user uses his SIG token to sign a newly compiled firmware: compilation and signing must be performed in a **secure environment** using a **disconnected and trusted PC host**. The signed firmware can be deployed using the DFU mode of WooKey in an unsafe environment, with the user authenticating with his DFU token.

2.6. Product environment description

The product can be used on typical workstations and laptops with no restrictions.

The product does not protect user data confidentiality when the **full authentication sequence** has been performed: a fully unlocked WooKey exposes such decrypted data to the host, and a malware on the host is then able to transparently read them.

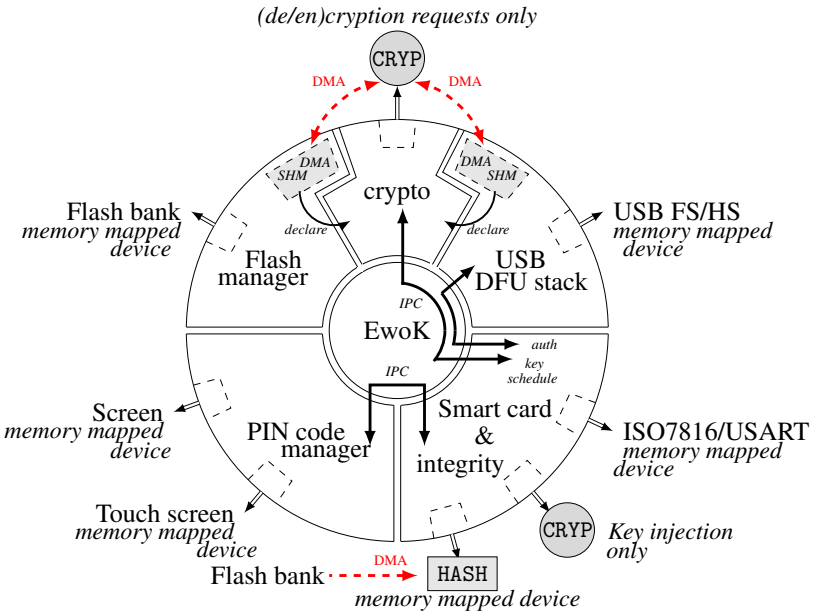

Figure 7: Software architecture – WooKey *DFU mode*

While user data at rest confidentiality is covered, user data integrity is not protected. The user **must** use complementary means to check his data integrity, e.g. using a file system with such features implemented.

In the WooKey life-cycle, the out-of-the-box and non-configured device must be handled in a **safe environment**.

2.7. Product typical users description

The only user that interacts with the product is the final user. He has therefore all rights on the platform, meaning he can configure it, use it, flash it, upgrade it, sign new firmwares, etc.

Typical users are those who care about **data confidentiality** and **trusted USB devices**. Employees of firms handling sensitive or confidential data, and traveling with them. Firms/administrations that use computers where only trusted devices must be plugged in. Military users who can loose their devices in the field.

2.8. Product evaluation perimeter

The evaluation perimeter concerns the entire product, except the hardware and platform part of the External Token (EAL 4 certified, see [27]). The embedded Javacard applets are part of the scope: only the underlying IC, native cryptographic libraries and Javacard embedded OS are not since they have been evaluated under the Common Criteria scheme.

The trusted platform (PC) on which firmwares are produced (compiled) and signed is also out of the scope of the evaluation.

3. Environment hypothesis description

[H1]. WooKey properly configured

We consider the WooKey device as correctly configured and locked, namely:

- The SoC RDP2 (ReadOut Protection level 2) flash lock is active through the proper **eFuses** (via dedicated software *not included* in the Tataouine SDK).
- The WooKey bootloader RDP2 check option is **enabled** (via the compilation configuration file).
- The WooKey bootloader flash lock option is **enabled** (via the compilation configuration file).
- The WooKey bootloader firmware header CRC32 check option is **enabled** (via the compilation configuration file).
- The WooKey kernel *reboot on panic* option is **enabled** (via the compilation configuration file).
- The serial debug console is **disabled** for both bootloader and kernel (via the compilation configuration file).

[H2]. Using three dedicated Javacard tokens

We suppose that the user is using the *high security profile* of the WooKey product (configurable using Tataouine). Namely, three distinct physical tokens are used for each token role:

- AUTH: the user authentication token used for WooKey platform unlocking in *nominal mode*.
- DFU: the user token used for WooKey platform unlocking in *DFU mode*, and for updates.
- SIG: the token used on the firmware production platform for firmware signature.

[H3]. Javacard tokens properly configured, locked with PIN, with a reasonable maximum PIN tries (typically 3)

We suppose that the Javacards are *locked*: the (default) global platform keys are modified with secret values and using third party tools. In the WooKey security context, attackers without knowledge of the secret GP keys must not be able to install new applets on the smart cards.

[H4]. Complex password

The user has set its own Pet Pin, Pet Name and User Pin using reasonable complex content. These information are not known from the attacker.

[H5]. Tokens

The user only holds the AUTH token for everyday usage. DFU token is kept safe and only used when updating the platform. SIG token is kept safe and only used **in a safe context, on a unaltered safe PC host** when signing firmware.

[H6]. Access

The attacker has a full access to the open source and open hardware data, is able to reproduce a board, firmwares and flash new Javacards (AUTH/DFU/SIG tokens). The attacker has a full physical access to the device.

4. Sensitive assets description

[A1]. User data

Stored in the microSD card. (De/En)crypted using the Master Secret Key MSK [A9].
Security need: **confidentiality**

[A2]. UserPIN

The UserPIN is used to fully unlock the data storage private key MSK [A9] stored in the Javacard.
Security need: **confidentiality**

[A3]. PetPIN

The PetPIN is used to initiate the mutual authentication between the device and the Javacard token.
Security need: **confidentiality**

[A4]. PetName

Personal passphrases which allows to *identify* the tokens (each token has its passphrase). It is printed on the screen and proves that the mutual authentication of both devices worked and that the Javacard is not tempered with.
Security need: **confidentiality, integrity**

[A5]. Derived Key (DK)

Key derived from the PetPIN [A3] using the PBKDF2 algorithm. It is used by the tokens to decrypt the Encrypted Local Key ELK [A6].
Security need: **confidentiality**

[A6]. Encrypted Local Key (ELK)

Key stored on the token. It is the encrypted form of the KPK [A7] using the Derived Key DK [A5].
Security need: **confidentiality, integrity**

[A7]. Key Platform Key (KPK)

Key computed by the token from the ELK (Encrypted Local Key) [A6] using a key derived from the PetPIN [A3]. It is sent to the platform after decryption to decrypt the Platform Keys PK [A8].
Security need: **confidentiality**

[A8]. Platform Keys (PK)

Keys stored ciphered on the platform by the Key Platform Key KPK [A7] and used by the platform to establish the secure channel with the token. PK include both sensitive assets: the ECDSA private key of the platform to establish the secure channel (must remain confidential and non tampered with), the ECDSA firmware verification public key (must remain non tampered with).
Security need: **confidentiality, integrity**

[A9]. Master Secret Key (MSK)

This key is the master secret key used to encrypt user data on the microSD card. It is stored in the AUTH token, and sent to the platform the token is fully unlocked (UserPIN [A2] and PetPIN [A3] presented).
Security need: **confidentiality, integrity**

[A10]. Key HMAC Device Firmware Update (KHDFU)

Key stored in the DFU and SIG tokens, used to verify the integrity of the update header by the use of an HMAC.
Security need: **confidentiality, integrity**

[A11]. Key Derivation Device Firmware Update (KDDFU)

Key Derivation Device Firmware Update (KDDFU), used to derive firmware chunk encryption and decryption. This key is stored in the DFU and SIG tokens.
Security need: **confidentiality, integrity**

[A12]. Derived Decryption Keys (DK_i)

These keys are derived by the DFU and SIG token using the Key Derivation Device Firmware Update KDDFU [A11]. The derivation mechanism encrypts a counter `IV+i` depending on the chunk number `i` to produce DK_i.
Security need: **confidentiality**

[A13]. Tokens ECDSA Secure Channel Private Keys

Each token (AUTH/DFU/SIG) has an ECDSA key pair used for secure channel establishment.
Security need: **confidentiality, integrity**

[A14]. SIG token ECDSA Firmware Signature Private Key

The SIG token stores the ECDSA firmware signature key used to sign firmwares.
Security need: **confidentiality, integrity**

5. Threats description

5.1. Attackers profiles

We consider that the adversary has logical and/or physical access to the device. He may try to read the data by connecting the device to a host or by reading the mass storage cells when the device is lost or stolen (so-called data at rest). We also consider physical tampering with the internal storage, firmware, or any other component present on the actual device. Logical attacks are in the scope, e.g. when connecting the device to an untrusted host that exploits software stacks vulnerabilities (USB stack, etc.), or by abusing any interface with malformed data. Our threat model also captures pre-authentication (i.e. before legitimate users authenticates to the device) side channel and fault injection attacks.

We **do not consider** threats where the legitimate user leaves his WooKey device connected to a PC host and **fully unlocked**. Whenever a legitimate user leaves his device, he must voluntarily **lock** the device, either by ejecting the token (only ejecting it is sufficient, no need to fully remove it), or by pressing the `lock` button on the authenticated touch screen GUI. The consequence of either of these actions is a reset of the WooKey device, necessitating a full unlocking whenever the user needs to access his data again.

5.2. Threats

[T1]. Data at rest

With a logical or physical access to the TOE, the attacker is able to retrieve the stored data.

[T2]. Persistent firmware corruption

The attacker corrupts the firmware with either a software or a hardware mean in order to create a persistent modification on the flash memory. This allows him to get access to all data stored or received by the device.

[T3]. Token cloning

The attacker clones a valid token making him able to authenticate with a platform.

[T4]. Token theft

The attackers steals a valid token allowing him to interact with it. He is thus able to understand how it works and also retrieve information that it stores.

[T5]. Device theft

The attacker steals a valid device allowing him to interact with it. He is thus able to understand how it works and also to retrieve information that it stores.

[T6]. Device cloning

The attackers clones a valid device making him able to have an authenticate platform, using the same secrets as the original device.

[T7]. Pre-authentication leak exploitation

The attacker is able to exploit a pre-authentication leak through side channel attacks to retrieve sensitive assets.

[T8]. Device upgrade exploitation

The attacker exploits a weakness in the device upgrade state-machine (unauthenticated firmware, corrupted firmware image injection, a race condition, etc.) in order to load a malicious firmware. This allows him to get access to all data stored or received by the device.

[T9]. Memory access

With a logical access the attacker gets access to the memory of the TOE to retrieve all loaded data or key.

[T10]. Communication spying

The attacker probes the communication between the token and the platform in order to gather all data that is transferred, and infers sensitive information/ways to perform further attacks.

[T11]. Firmware Downgrade

The attacker is able to downgrade the firmware version (anti-rollback bypass) by loading one that contains a known vulnerability. He is then able to retrieve all data the of the device.

[T12]. Debug Access

The attacker takes advantage of the debug access available on the device to retrieve the data from RAM or flash which may contain sensitive information.

[T13]. Malicious update

The attacker updates the firmware with a malicious firmware in order to steal the user's related data.

6. Security functions description

[SF1]. PetPIN pre-authentication phase

Before unlocking the device storage, the user enters a first PIN, denoted PetPIN [A3], in order to initiate the mutual authentication procedure between the smart card and the device. The number of mutual authentication failures is limited by a counter stored in the smart card. The PetPIN try counter is also limited in the smart card, however when the user enters his PetPIN PBKDF2 is performed with the token to derive DK [A4] and decrypt ELK. The PetPIN is really sent to the token only once a secure channel is mounted, a in this case a PetPIN failure counter is decremented if the PetPIN is not OK.

[SF2]. PetName user validation

At mission preparation time, a pass phrase is set in the smart card internal memory. This passphrase is returned by the token to the device if the mutual authentication succeeds and is printed out onscreen. If this passphrase does not correspond to the one the user has previously set, the user is able to detect that something went wrong (the device or the smart card has been tampered with). The user must stop using the device usage immediately.

[SF3]. User authentication and MSK unlock

When [SF1] and [SF2] succeed, the user enters the User PIN. This PIN allows to unlock the smart card MSK key [A8]: MSK is then transmitted to the device and injected in the STM32 `CRYP` cryptographic coprocessor.

[SF4]. Secure channel

Excluding the PBKDF2 derived PetPIN, any communication between the device and the smart card is transferred through a secure channel over the ISO716-3 line. This secure channel supports anti-replay protection and ensures integrity and confidentiality of the transmitted data.

[SF5]. User data encryption

User data is stored in the device's microSD. They are encrypted using the AES-CBC-ESSIV algorithm, using a 256 bits key (the MSK [A8]). User data at rest integrity is not supported.

[SF6]. Hardened pre-authentication cryptographic actions

The overall cryptographic algorithms execution made before the complete authentication sequence is protected against side channel attacks.

[SF7]. Device platform keys PK confidentiality

The device platform keys PK, used to authenticate the device against the smart card during the first step of the authentication phase, are stored encrypted and can be decrypted only in association with the PetPIN and the smart card as a cryptographic oracle.

[SF8]. Firmware resilience

In order to support upgrade corruption (voluntary or not) the firmware is duplicated and the upgraded firmware is always the one not currently used. The newly updated firmware is activated only after a full integrity check and signature validation in association with the authenticated firmware header received during the upgrade sequence.

[SF9]. Differentiated upgrade mode and role

Upgrading the firmware requires a dedicated DFU token, which is separated from the mass storage, nominal user, AUTH token. The complete key chain (including platform keys, smart card keys, and so on) are segregated. The complete PIN chain (including PetPIN, userPIN and PetName) are also segregated. Compromising the AUTH (resp. the DFU) token should not yield in compromising the AUTH token.

[SF10]. Voluntary firmware upgrade mode

Booting to the DFU mode to perform a firmware upgrade is a voluntary action since it requires a user button push at boot time.

[SF11]. Firmware authentication

In order to ensure the firmware authenticity, an ECDSA signature is computed with a private key enclosed in the SIG token on a trusted dedicated host. The signature verification is performed using the ECDSA public key embedded in the WooKey platform internal flash (part of the Encrypted Local Keys ELK [A5]). This verification is performed during DFU after the firmware is written in flash.

[SF12]. Firmware upgrade anti-rollback

The firmware upgrade sequence checks the firmware version in the authenticated header. The version must be greater than the current one to avoid any rollback upgrade (e.g. to exploit a vulnerability in an old firmware).

[SF13]. Least privilege principle

The firmware relies on a microkernel architecture (EwoK) to enforce the least privilege principle by isolating the drivers in their own address space. Drivers are implemented as user tasks with limited access rights, and dangerous operations (such as DMA transfers) are subject to microkernel checks enforcement.

[SF14]. MPU usage

The MPU is a programmable unit that allows privileged software, often a kernel, to define memory access permissions in order to isolate memory regions. EwoK uses this mechanism to enforce confinement and privilege separation between tasks executed in user mode, so that they can't break out of their address space.

[SF15]. Debug access deactivated

JTAG port is deactivated. The flash internal memory is set in RDP2 mode, excluding debugger access, any external read or write access to neither flash nor RAM memory.

References

[1] The manifest covering the firmware version evaluation. `https://github.com/wookey-project/manifest/blob/master/soft/snapshots/wookey_20190903.xml`, 2019.

[2] The wookey project documentation. `https://wookey-project.github.io/`, 2019.

[3] The wookey project github repository. `https://github.com/wookey-project`, 2019.

[4] The wookey project main board pcb. `https://github.com/wookey-project/hard-wookey-motherboard`, 2019.

[5] The wookey project screen board pcb. `https://github.com/wookey-project/hard-wookey-screen`, 2019.

[6] ADACORE. GnatProve formal verification tool. `https://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_run_gnatprove.html`.

[7] ALTRAN, ADACORE. SPARK programming language. `https://www.spark-2014.org/`.

[8] BRANDON, C., AND CHAPIN, P. The Use of SPARK in a Complex Spacecraft. *ACM SIGAda Ada Letters 36*, 2 (2017), 18–21.

[9] CHEN, Y., ZHANG, Y., WANG, Z., AND WEI, T. Downgrade Attack on TrustZone. *arXiv preprint arXiv:1707.05082* (2017).

[10] COMMITTEE, T. T. Scsi architecture. `http://www.t10.org/scsi-3.htm`.

[11] DEEG, M., AND SEBASTIAN, S. *Cryptographically Secure? SySS Cracks a USB Flash Drive*, 2009.

[12] DMITRY NEDOSPASOV, JOSH DATKO, T. R. Hardware Wallet Vulnerabilities. `https://media.ccc.de/v/35c3-9563-wallet_fail/oembed`, 2018.

[13] DÜRMUTH, M., GÜNEYSU, T., KASPER, M., PAAR, C., YALÇIN, T., AND ZIMMERMANN, R. Evaluation of standardized password-based key derivation against parallel processing platforms. In *ESORICS 2012* (2012).

[14] DWORKIN, M. NIST SP 800-38E, Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices. `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38e.pdf`, 2010.

[15] FRUHWIRTH, C. *New Methods in Hard Disk Encryption*. na, 2005.

[16] GERLINSKY, C. Breaking Code Read Protection on the NXP LPC-family Microcontrollers. `https://recon.cx/2017/brussels/resources/slides/RECON-BRX-2017-Breaking_CRP_on_NXP_LPC_Microcontrollers_slides.pdf`, 2017.

[17] GÖTZFRIED, J., AND MÜLLER, T. Analysing android's full disk encryption feature. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications 5* (03 2014), 84–100.

[18] HENRY, T., RIVENBURG, D., AND STIRLING, D. Universal Serial Bus Device Class Specification for Device Firmware Upgrade. *Aug 5* (2004), 47.

[19] HOLDINGS, A. ARMv7-M Architecture Reference Manual. `https://static.docs.arm.com/ddi0403/e/DDI0403E_d_armv7m_arm.pdf`, 2010.

[20] HOTZ, G., AND TEAM, C.-D. Limera1n exploit (souce code). `https://github.com/Chronic-Dev/syringe/blob/master/syringe/exploits/limera1n/limera1n.c`, 2010.

[21] KREDER, K. Hardware Wallet Vulnerabilities. `https://blog.gridplus.io/hardware-wallet-vulnerabilities-f20688361b88`.

[22] LELL, J. Practical malleability attack against CBC-Encrypted LUKS partitions, 2013.

[23] MELTEM SÖNMEZ TURAN, ELAINE BARKER, WILLIAM BURR, AND LILY CHEN. Recommendation for Password-Based Key Derivation, Part 1: Storage Applications. NIST Special Publication 800-132, National Institute of Standards and Technology, 2010.

[24] PAUL A. GRASSI, JAMES L. FENTON, ELAINE M. NEWTON, RAY A. PERLNER,ANDREW R. REGENSCHEID, WILLIAM E. BURR,JUSTIN P. RICHER. Digital Identity Guidelines, Authentication and Lifecycle Management. NIST Special Publication 800-63B, National Institute of Standards and Technology, 2017.

[25] RUIZ, J. F. Going real-time with Ada 2012 and GNAT. *ACM SIGAda Ada Letters 33*, 1 (2013), 45–52.

[26] SAHAMI SHIRAZI, A., MOGHADAM, P., KETABDAR, H., AND SCHMIDT, A. Assessing the vulnerability of magnetic gestural authentication to video-based shoulder surfing attacks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2012), ACM, pp. 2045–2048.

[27] SEMICONDUCTORS, N. Nxp j3d081_m59_df, and j3d081_m61_df secure smart card controller revision 2. `https://www.commoncriteriaportal.org/files/epfiles/0860b_pdf.pdf`, 2013.

[28] TEMKIN, K., AND SZEKELY, M. Fusée Gelée Exploit (souce code). `https://github.com/Cease-and-DeSwitch/fusee-launcher`, 2018.

A Mappings

1.1. Assets - Threats

Table 1: Mapping Assets - Threats

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
A1	X	X			X	X	X	X	X		X	X	X
A2		X						X	X	X	X	X	X
A3		X	X					X	X		X	X	X
A4	X	X	X					X	X		X	X	X
A5		X						X	X	X	X	X	X
A6			X	X									
A7		X						X	X	X	X	X	X
A8	X	X						X	X		X	X	X
A9		X	X	X				X	X	X	X	X	X
A10			X	X									
A11			X	X									
A12		X	X	X				X	X	X	X	X	X
A13			X	X									
A14			X	X									

X: applicable only to the AUTH token
X: applicable only to the DFU and SIG tokens
X: applicable only to all tokens
X: applicable only to the SIG token

1.2. Threats - Security Functions

Table 2: Mapping Threats - Security Functions

	SF1	SF2	SF3	SF4	SF5	SF6	SF7	SF8	SF9	SF10	SF11	SF12	SF13	SF14	SF15
T1	X		X		X		X								
T2								X	X		X				
T3	X														
T4	X						X								
T5	X	X	X				X								
T6	X														
T7						X									
T8								X	X		X				
T9													X	X	X
T10				X											
T11												X			
T12															X
T13								X	X	X	X				

B Details on cryptography used in WooKey

2.1. Keys/assets generation

All the keys (with the exceptions described hereafter) implied in WooKey's cryptography are **generated on the trusted host PC** that compiles and signs the firmwares using `/dev/random` source. It is the end user's responsibility to ensure proper entropy from this source (e.g. by feeding Linux entropy pool with a smart card AIS31 certified TRNG random source).

The ephemeral session keys described in Appendix.2.2. are generated through the tokens AIS31 certified TRNG on one side, and through the STM32 non certified TRNG on the other side (see Appendix.2.5.).

The PetPINs, UserPINs and PetNames are chosen by the user. It is the end user's responsibility to choose non trivial PINs and PetName.

2.2. The Secure Channel with the tokens

The secure channel is established between any token (AUTH/DFU/SIG) and the platform or the PC (communication can be performed on a regular PC through PCSC for instance).

2.2.1. Secure channel core

The secure channel makes use of an **authenticated ephemeral ECDH** protocol that derives session keys.

In order to establish the secure channel, an ECDSA key pair is used for the platform denoted $ECDSA_{priv}^{platform}$ and $ECDSA_{pub}^{platform}$ respectively for the platform private and public key (they are parts of asset PK [A8]). In our case, the platform is either WooKey or a host PC.

Another ECDSA key pair is used for the token, denoted $ECDSA_{priv}^{token}$ and $ECDSA_{pub}^{token}$ respectively for the token private and public key (see asset [A13]). The token can be either of AUTH, DFU or SIG type.

ECDSA is used with SHA-256 hash function over a choice of one of three possible curves: FRP256V1, NIST SECP256R1 and BRAINPOOLP256R1.

The secure channel establishment supposes a master (initiator) and a slave (receiver). In the case of WooKey, the master is always the platform or host PC, and the slave is always the token. Whenever a secure channel must be established, the master randomly chooses a private scalar d and computes $Q = d \times G$ where G is the generator of the elliptic curve. The master then sends over the ISO7816-3 line the public point Q and the ECDSA signature of Q: $Q||ECDSA - SHA256(ECDSA_{priv}^{platform}, Q)$. Q is made of projective coordinates with $Q_z = 1$, i.e. the equivalent affine coordinate, and the encoding is simply the concatenation of the three coordinates $Q = Q_x||Q_y||Q_z$ with each coordinate being a big endian big number on 256 bits.

The slave receives $Q||ECDSA - SHA256(ECDSA_{priv}^{platform}, Q)$, checks the signature using the public key $ECDSA_{pub}^{platform}$, randomly draws d' and computes $Q'' = d' \times Q = d' \times (d \times G)$. The slaves then sends overt the ISO7816-3 line $Q'||ECDSA - SHA256(ECDSA_{priv}^{token}, Q')$ with $Q' = d' \times G =$

$Q'_x||Q'_y||Q'_z$ with each coordinate being a big endian big number on 256 bits. The master receives this, checks the signature using $ECDSA^{token}_{pub}$, and computes $Q'' = d \times Q' = d \times (d' \times G)$.

Now both the master and slave have $Q'' = Q''_x||Q''_y||Q''_z$ with $Q''_z = 1$ to have the unique affine representation of the point. Each party can compute a session key for encryption, a session key for integrity, and an IV (Initialization Vector) for anti-replay:

- Encryption session key of 16 bytes (for AES-128-CTR): computed using $SHA256('AES\_SESSION\_KEY'||Q''_x)$, and keeping the first 16 bytes of the result, with $'AES\_SESSION\_KEY'$ being a constant diversification string.
- Integrity session key of 32 bytes (for HMAC-SHA-256): computed using $SHA256('HMAC\_SESSION\_KEY'||Q''_x)$, with $'HMAC\_SESSION\_KEY'$ being a constant diversification string.
- IV of 16 bytes: computed using $SHA256('SESSION\_IV'||Q''_x)$, and keeping the first 16 bytes, with $'SESSION\_IV'$ being a constant diversification string.

The secure channel's purpose is to ensure confidentiality and integrity of data transiting between the two trusted peers (the master and the slave). Elements that transit between these peers are ISO7816 APDUs, and they are subject to some constraints imposed by the ISO781 protocol. Namely, the format of an APDU is the following:

`[CLA] [INS] [P1] [P2] [Lc] [PAYLOAD]`

With `[Lc]` being the length of the payload `[PAYLOAD]` (we only deal with short APDU format in the case of WooKey secure channel). Because of their implication in the ISO7816, `[CLA]`, `[INS]`, `[P1]`, `[P2]` and `[Lc]` cannot be easily encrypted. Since they do not convey confidential data in the case of WooKey, this is not an issue. They are however covered by the APDU integrity check (see below). The secure channel uses an **encrypt-then-MAC** paradigm. The responses from the slave have the following form:

`[ANSWER_PAYLOAD] [SW1] [SW2]`

The slave's answers encryption and MAC follow the same logic as for the master's commands, with `[SW1]` and `[SW2]` being strongly bound to the ISO7816 protocol and hence not encrypted (but covered by the MAC integrity).

APDU encryption: `[PAYLOAD]` is encrypted using the secure channel encryption key, AES-128-CTR and an IV equal to the current value of the IV (fresh random IV from ECDH ensures no reuse of old IVs):

APDU integrity: the integrity of the whole APDU is computed using HMAC-SHA256 with a prepended IV value for anti-replay on the encrypted payload (encrypt-then-MAC):

$$Tag\_command = HMAC - SHA256(IV||CLA||INS||P1||P2||Lc||enc(PAYLOAD)$$

The slave's answer MAC is computed as follows:

$$Tag\_answer = HMAC - SHA256(SW1||SW2||enc(ANSWER\_PAYLOAD))$$

The MAC tags are appended to the encrypted payload, and the APDU is reformatted (with conforming `Lc` that takes into account the MAC) to be conforming to the ISO7816 protocol.

IV handling: for each APDU, IV is incremented by the number of encrypted 16 bytes AES blocks (with `PAYLOAD` of size strictly less than a block counting for zero). In order to cover payloads strictly smaller than an AES block (including empty payloads), the IV is also always incremented by one.

Session keys update: in order to bind the secure channel with the PetPIN and UserPIN, the channel encryption and MAC keys are diversified whenever the user enters a PIN. When the PIN is entered by the user, it is padded to a 16 bytes length with the last byte corresponding to its real size:

$$Padded\_PIN = PIN\ ||00||\ ...||00||len(PIN)$$

The original PIN length is strictly less than 16 bytes (this is a hard limitation in the WooKey SDK and platform). The secure channel keys diversification use the following scheme (where *enc_key* is the original AES-128-CTR encryption key, and *hmac_key* is the original HMAC key):

$$Diversified\_enc\_key = enc\_key \oplus SHA256(Padded\_PIN||\ IV)[0:16]$$
$$Diversified\_hmac\_key = hmac\_key \oplus SHA256(Padded\_PIN||\ IV)$$

Here, IV is the current value of the IV.

Sensitive data (over-)encryption: in addition to the regular encryption of the secure channel, some sensitive data transiting inside the secure channel are also encrypted using a key derived from the UserPIN and the first IV (i.e. the IV derived from the ECDH value Q''_x). Hence, let $First\_IV = SHA256('SESSION\_IV'||Q''_x)$. The derived key is computed from the original (unpadded) UserPIN as follows:

$$Sensitive\_data\_key = SHA256(First\_IV||\ SHA256(UserPIN))$$

This key is used as an AES-128-CBC encryption key, with the current secure channel IV as Initialization Vector. There is no padding issue here since the sensitive data that are encrypted are ensured (by choice/design) to be aligned on an AES 16 bytes block size. In WooKey, the sensitive data that is encrypted using this scheme is the Master Secret Key (MSK, asset [A9]).

2.2.2. Local keys (enc/dec)ryption

The WooKey platform contains two pairs of ECDSA keys for the secure channel, one for the nominal mode and one for the DFU mode. Each ECDSA key pair is used in conjunction with the associated token (AUTH or DFU). The AUTH ECDSA key pairs are denoted $ECDSA^{AUTH\_plat}_{priv}$ and $ECDSA^{AUTH\_plat}_{pub}$ for the platform side, and $ECDSA^{AUTH\_token}_{priv}$ and $ECDSA^{AUTH\_token}_{pub}$ for the AUTH token side. The DFU ECDSA key pairs are denoted $ECDSA^{DFU\_plat}_{priv}$ and $ECDSA^{DFU\_plat}_{pub}$ for the platform side, and $ECDSA^{DFU\_token}_{priv}$ and $ECDSA^{DFU\_token}_{pub}$ for the DFU token side.

The host PC that is used to sign firmwares uses a dedicated ECDSA key pair. These keys are denoted $ECDSA_{priv}^{SIG\_plat}$ and $ECDSA_{pub}^{SIG\_plat}$. On the SIG token side, the corresponding key pair is denoted $ECDSA_{priv}^{SIG\_token}$ and $ECDSA_{pub}^{SIG\_token}$.

In addition to secure channel associated keys, the firmware **signature** key pair is denoted $ECDSA_{priv}^{Firm}$ and $ECDSA_{pub}^{Firm}$. The key pair is enclosed solely in the SIG token, and the public part $ECDSA_{pub}^{Firm}$ is present in the DFU platform to check the signature during DFU updates.

Platform nominal mode/AUTH token:

On the platform, in **nominal mode**, $ECDSA_{priv}^{AUTH\_plat}$, $ECDSA_{pub}^{AUTH\_plat}$ and $ECDSA_{pub}^{AUTH\_token}$ are stored encrypted and integrity protected. They correspond to asset [A8] for the nominal token and mode. The encryption makes use of AES-128-CTR with a random `IV` that is denoted $plat\_AUTH\_IV$ in the following. Let the encrypted platform keys be:

$$EPK^{AUTH} = AES-128-CTR(plat\_AUTH\_IV, ECDSA_{pub}^{AUTH\_token} || ECDSA_{priv}^{AUTH\_plat} || \ ECDSA_{pub}^{AUTH\_plat})$$

using the AUTH Key Platform Key KPK (asset [A7]). The AUTH KPK is enclosed inside the AUTH token flash, it is 512-bit long and is made of a 128-bit AES-128-CTR key (first 128 bits of the 512 bits) and a 256-bit HMAC-SHA256 key (last 256 bits of the 512 bits).

The HMAC tag is computed on the following data using the HMAC-SHA256 key part of the AUTH KPK:

$$Tag^{AUTH} = HMAC-SHA-256(plat\_AUTH\_IV || plat\_AUTH\_SALT || \ EPK^{AUTH})$$

Then, the following data are stored in the internal flash of the platform (inside the firmware):

$$plat\_AUTH\_IV || \ plat\_AUTH\_SALT || \ Tag^{AUTH} || \ EPK^{AUTH}$$

Platform DFU mode/DFU token:

On the platform, in **DFU mode**, $ECDSA_{priv}^{DFU\_plat}$, $ECDSA_{pub}^{DFU\_plat}$, $ECDSA_{pub}^{DFU\_token}$ and $ECDSA_{pub}^{Firm}$ are stored encrypted and integrity protected. They correspond to asset [A8] for the DFU token and mode. The encryption makes use of AES-128-CTR with a random `IV` that is denoted $plat\_DFU\_IV$ in the following. Let the encrypted platform keys be:

$$EPK^{DFU} = AES-128-CTR(plat\_DFU\_IV, ECDSA_{pub}^{DFU\_token} || ECDSA_{priv}^{DFU\_plat} || \ ECDSA_{pub}^{DFU\_plat} \ || \ ECDSA_{pub}^{Firm})$$

using the DFU Key Platform Key KPK (asset [A7]). The DFU KPK is enclosed inside the DFU token flash, it is 512-bit long and is made of a 128-bit AES-128-CTR key (first 128 bits of the 512 bits) and a 256-bit HMAC-SHA256 key (last 256 bits of the 512 bits).

The HMAC tag is computed on the following data using the HMAC-SHA256 key part of the DFU KPK:

$$Tag^{DFU} = HMAC-SHA-256(plat\_DFU\_IV || plat\_DFU\_SALT || \ EPK^{DFU})$$

Then, the following data are stored in the internal flash of the platform (inside the firmware):

$$plat\_DFU\_IV || plat\_DFU\_SALT || Tag^{DFU} || EPK^{DFU}$$

PC host for firmware signature/SIG token:

On the PC host platform, $ECDSA_{priv}^{SIG\_plat}$, $ECDSA_{pub}^{SIG\_plat}$, $ECDSA_{pub}^{SIG\_token}$ and $ECDSA_{pub}^{Firm}$ are stored encrypted and integrity protected. They correspond to asset [A8] for the SIG token and mode. The encryption makes use of AES-128-CTR with a random `IV` that is denoted $plat\_SIG\_IV$ in the following. Let the encrypted platform keys be:

$$EPK^{SIG} = AES-128-CTR(plat\_SIG\_IV, ECDSA_{pub}^{SIG\_token} || ECDSA_{priv}^{SIG\_plat} || ECDSA_{pub}^{SIG\_plat} || ECDSA_{pub}^{Firm})$$

using the SIG Key Platform Key KPK (asset [A7]). The SIG KPK is enclosed inside the SIG token flash, it is 512-bit long and is made of a 128-bit AES-128-CTR key (first 128 bits of the 512 bits) and a 256-bit HMAC-SHA256 key (last 256 bits of the 512 bits).

The HMAC tag is computed on the following data using the HMAC-SHA256 key part of the SIG KPK:

$$Tag^{SIG} = HMAC-SHA-256(plat\_SIG\_IV || plat\_SIG\_SALT || EPK^{SIG})$$

Then, the following data are stored in the internal flash of the platform (inside the firmware):

$$plat\_SIG\_IV || plat\_SIG\_SALT || Tag^{SIG} || EPK^{SIG}$$

Computing the (AUTH, DFU, SIG) KPK for local keys decryption:

The Key Platform Key KPK (for each token AUTH/DFU/SIG) is made of 512 bits. A 128-bit AES key used for AES-128-CTR to decrypt the EPK local platform keys is extracted from the first 128 bits, and a 256-bit HMAC-SHA256 key is extracted from the last 256 bits. This key is enclosed **encrypted** in the corresponding token and provisioned (encrypted) during the manufacturing phase of WooKey. It is encrypted with the Derived Key DK (asset [A5]) using AES-128-ECB on the four 128-bit parts of the KPK, knowing that DK is also 512 bits and made of four 128 bits sub-keys.

$$DK = DK^0 || DK^1 || DK^2 || DK^3, \text{ four 128-bit parts}$$

$$KPK = AES-128-ECB_{DK^0}(EPK^0) || AES-128-ECB_{DK^1}(EPK^1) || AES-128-ECB_{DK^2}(EPK^2) || AES-128-ECB_{DK^3}(EPK^3)$$
with AES decryption on four 128-bit parts.

The 512-bit Derived Key DK is computed from the PetPIN when it is entered by the user using PBKDF2-SHA512, with a salt of 128 bits provided by the $platform\_XXX\_SALT$ (XXX being AUTH, DFU or SIG depending on the context), and with 4096 iterations.

In order to limit brute-force attacks and given the fact that 4096 iterations are quite low (imposed by the fact that our MCU horsepower is quite limited), the KPK key decryption timing in the token is incremented until a secure channel is properly mounted (meaning that the PetPIN and platform keys were OK).

2.3. DFU and Firmware encryption/signature details

The WooKey device makes use of encryption and signing for the firmware. Because it is complex for a MCU to store and check an encrypted firmware in its very limited SRAM, we have chosen to decrypt the firmware and then check the signature on the clear text in flash.

Encrypted and signed firmware generation: in the firmware signature case, we have two parties: a host PC that is the master, and the SIG token that is the slave. The two parties first mount a secure channel as described in Appendix.2.2., and then the user presents his UserPIN to fully unlock the SIG token features.

First of all, the PC computes the SHA256 value of the firmware to be encrypted and signed, and asks the token to sign this value using ECDSA-SHA256, meaning that the token replies with:

$$Firm\_sig = ECDSA - SHA256(ECDSA_{priv}^{Firm}, firmware)$$

where firmware is the firmware raw binary in clear text. Note that the double SHA256 hash is due to the Javacard ECDSA API limitation since this API does not offer raw signature (but only modes combined with a hash function).

The user can now ask the token to *generate a HMAC for the firmware header* by sending a firmware *header* denoted HDR_raw and containing a magic value on 4 bytes, the partition type (FLIP or FLOP) on 4 bytes, the length of the padding data in the header and before the payload on 4 bytes (this padding is mainly due to DFU protocol constraints), the signature length on 4 bytes, the firmware encryption chunk size on 4 bytes, and finally the signature $Firm\_sig$ previously computed on 64 bytes (this is the typical concatenation of r and s as a result of ECDSA). The token then responds $IV||HMAC - SHA256(\overline{HDR})$, with the HMAC computed using the Key HMAC Device Firmware Update KHDFU (asset [A10]) key (a 32 bytes dedicated key), and $\overline{HDR}$ being the header containing the randomly generated IV (see Figure 6 for the overview of the elements covered by the HMAC and the introduction of the IV). The full header HDR containing both the signature and the HMAC can then be composed by the host PC with the various pieces (metadata, signature, etc.).

After this, the PC is ready to perform the firmware encryption. For every chunk of firmware (that is split in equal part of maximum size of the chunk size sent to the token), a new encryption key DK_i (assets [A12]) is asked to the token by sending the value i, with DK_i being computed as follows:

$$DK_i = AES - 128 - CBC(IV + i)$$

with encryption key denoted $KDDFU\_ENC$, and with an IV denoted $KDDFU\_IV$ (the KDDFU asset [A11] is a 32 bytes long value split in two 16 bytes long values: first 16 bytes are $KDDFU\_ENC$, second 16 bytes are $KDDFU\_IV$).

These DK_i keys are used to encrypt the firmware chunks using AES-128-CTR and null IVs:

$$Encrypted\_firmware\_chunk_i = AES - 128 - CTR(firmware\_chunk_i)$$

Then, all the encrypted firmware chunks are concatenated together and the resulting binary is the exact same size of the original firmware thanks to the counter mode. The HMACed header HDR is padded to the next 4096 bytes boundary with a zero padding, and then the raw binary encrypted firmware is concatenated. The zero padding is useful because of the way the USB DFU protocol

truncates its own chunks, and because a header validation will be asked to the user before installing the update. This padding is not covered by any integrity check, but it is always skipped by the device when updating and never interpreted/parsed, so it should not induce security issues. The length of such padding is enclosed in the HMACed header *HDR* ensuring that no ambiguous skipping is performed when receiving a firmware.

Encrypted and signed firmware installation:

The user first establishes a secure channel using his PetPIN with the DFU token, confirms the PetName, and unlocks the DFU token with his UserPIN.

When the user sends a signed firmware using `dfu-util`, the first DFU chunk of 4096 bytes is sent to the device that pauses the DFU upload. The header is parsed, the padding is ignored, and the header HMAC is checked by interacting with the DFU token. This token contains the same KDDFU (asset [A11]) and KHDFU (asset [A10]) as the SIG token, and is able to open a decryption session and to derive the sessions keys DK_i.

Once the HMAC of the header *HDR* is checked, a strict anti-rollback check is performed, and the user is asked to validate the firmware through the touchscreen (the version and the magic of the firmware, recovered from the header, are printed). After hitting OK, the full firmware is sent and transparently decrypted through session keys injected in the platform hardware accelerator and AES-CTR running.

Whenever the decrypted firmware chunks have been deciphered in the STM32 internal flash, the firmware ECDSA signature verification is performed on the clear text flash content. Only when this signature is checked to be OK the hash value of the new firmware is copied in the boot partition, and the boot partition flag is atomically set to one.

After reboot, the bootloader checks the flag, and if it is OK the SHA256 hash of the partition to boot is computed and checked. Any error here yields in a reboot.

2.4. User data encryption details (AES-256-CBC-ESSIV)

User data on the SD card are encrypted using AES-256-CBC-ESSIV, with the master key MSK (asset [A9]) of 256-bit being stored inside the AUTH token, and only provided after a full unlocking of this token.

The CBC-ESSIV mode makes use of the hash value of MSK in order to compute the Initialization Vectors of the encryption of each sector:

$$ESSIV\_key = SHA256(MSK)$$

A sector here is a SCSI sector, configurable at firmware compilation from 512 bytes to 4096 bytes in WooKey. Hence, a new IV is derived for each sector from the sector number and a value derived from the SD card unique serial number (SD CID register value of 128 bits obtained through the SDIO command 10) using the ESSIV key:

$$Encrypted\_sector\_i = AES-256-CBC(sector_i)$$
with $ESSIV\_key$ as encryption key and:
$$IV = AES-256-ECB(encode32(i)||SD\_CARD\_ID)$$

With $encode32(i)$ being the big endian encoding of i on 32 bits, and:

$$SD\_CARD\_ID = SHA256(SD\_vendor\_ID||SD\_serial)[:12]$$

(i.e. first 96 bits of the SHA256 of the SD vendor ID concatenated with the serial number).

2.5. Random generation

The random generation makes use of the TRNG (True Random Number Generator) of the STM32 device. This TRNG is **not** AIS31 certified, and a PRNG should be used on top of it. However, this is not the case and this a future work in the WooKey project.